Project 5: Machine Learning - Enron Scandal

Tonima T Ananna

November 6, 2018

1 Introduction

The goal of this project is to find a way to distinguish between potential persons of interest (POI) involved in the Enron scandal, and normal Enron employees. We are given several important features about each person in our dataset, such as their salary, bonus, total stock, number of emails sent to and received from known persons of interest etc. From this information, there may be a feature or a combination of features that would help us identify parties who might have done something illegal. It might be much more complicated than just combining features, and we might need to deploy a machine learning algorithm to help us identify POIs.

A description of all the functions in poi_id.py is given at the end.

2 Outliers

To investigate the dataset, I wrote a few functions that would allow me to look up people with salaries/bonuses above some particular limit (find_an_interesting_person()). I wrote these functions based on necessity while investigating the dataset. There is a function that allows us to plot any three features at a time (compare_three_features()), and it seemed interesting to be able to look up the names of some extreme outliers. Also, another function allows us to look up all the information with the name of the person (find_by_name()).

To investigate outliers, I used this 3D plot, as well as a function that takes in any feature, and finds the first and third quartile value (Q1 and Q3 respectively), and the interquartile range (IQR). Then outliers are defined as values that lie outside this range: [Q1 - 1.5 $\times$ IQR, Q3 + 1.5 $\times$ IQR]. If this range is too extreme (below the minimum or maximum value for that feature), then this function just prints out the top and bottom 5% values for that feature, along with the names of the corresponding persons. By looking at the printout, we can judge whether this number truly is an outlier that should be ignored, or a legitimate value or even a known person of interest.

By looking at these values, I think the only real outlier I found is the "TOTAL" column - which is removed. Another interesting candidate I noticed in the enroninsiderpay pdf is The Travel Agency in the Park - which is not a person, but it can be an entity of interest, so I kept it. A lot of values become 0 because no information is available, and I thought about whether or not to ignore entries that have "NaN" features, but then we will end up ignoring too much, and I think that ensemble methods can still identify POIs if some features are missing, if enough weak estimators identifies someone as a possible POI. Also, I don't throw away extremely high "outlier" values because they are often POIs, and we already have few POIs to train the algorithm so we will lose important data if we get rid of them.

3 Number of NaNs in each field

After find_outliers function also lists the total number of NaNs in a field. There are 145 entries (after excluding TOTAL, as described in the previous section) in the database, 18 pois and 127 non-pois. The numbers of NANs in each class are:

- salary: poi nan - 1/18 = 5%, non poi nan - 50/127 = 39%.
- total payments: poi nan - 0/18, non poi nan - 21/127.
- bonus: poi nan - 2/18 = 11.1%, non poi nan - 62/127 = 48.8%.

- deferred income: poi nan - 7/18, non poi nan - 90/127.
- total stock value: poi nan - 0/18, non poi nan - 20/127.
- expenses: poi nan - 0/18, non poi nan - 51/127.
- exercised stock options: poi nan - 6/18, non poi nan - 38/127.
- long term incentive: poi nan - 6/18, non poi nan - 74/127.
- restricted stock: poi nan - 1/18, non poi nan - 35/127.
- director fees: poi nan - 18/18, non poi nan - 111/127.
- to messages: poi nan - 4/18, non poi nan - 55/127.
- from poi to this person: poi nan - 4/18, non poi nan - 55/127.
- from messages: poi nan - 4/18, non poi nan - 55/127.
- from this person to poi: poi nan - 4/18, non poi nan - 55/127.
- shared receipt: poi nan - 4/18, non poi nan - 55/127.

Director fees has a lot of nan values.

4 Feature Selection

I did a two step feature selection: **firstly**, I tried to look at how POIs/non-POIs are distributed in the 3D scatter plots of any three features (the initial version was a 2D plot), and checked how the features are correlated to each other using a r^2 correlation metric. Then I compared with SelecKbest feature selection.

The r^2 correlation metric doesn't work perfectly because of the “NaN” values which are converted to 0. I found that total payments and salary have a 55% r^2 correlation, but looks a lot more correlated in the plot. Also, ‘shared_receipt_with_poi’ and ‘from_poi_to_this_person’ has a 67% correlation. It makes sense not to use both features at once that are so correlated - but if one of these features is “NaN” the other can provide some information. I picked these 7 as the most promising features by eye:

feature_names=[“shared_receipt_with_poi”, ‘from_this_person_to_poi’, ‘from_poi_to_this_person’, ‘long_term_incentive’, “salary”, ‘exercised_stock_options’, “bonus”]

After carrying out this “organic feature selection”, I decided to check if my intuition about the best features were actually correct. So I used SelectKBest to get a direct scoring of the best features, and here is how they are scored:

bonus: 2.7306208, exercised stock options: 2.32966999, salary: 2.04040368, total stock value: 1.95336505, shared receipt with poi: 1.53635639, director fees: 1.44794604, from poi to this person: 1.39060327, from this person to poi: 1.07066958, expenses: 0.985433458, long term incentive: 0.504353443, total payments: 0.116074234, restricted stock: 0.191187977, to messages: 0.260304675, deferred income: 0.0443941224, from messages: 1.24324242e-05

There is a lot of overlap. I tried a simple decision tree to test different combination of best features according to SelecKBest + features I thought are promising and here are the results:

- (My initial intuition) ‘from_this_person_to_poi’, ‘long_term_incentive’, ‘salary’: Precision: 0.17368 Recall: 0.14850 F1: 0.16011 (terrible)
- ‘bonus’, ‘long_term_incentive’, ‘salary’ - Precision: 0.25211 Recall: 0.19400 F1: 0.21927 (better)
- ‘exercised_stock_options’, ‘long_term_incentive’, ‘salary’, ‘bonus’: Precision: 0.27720 Recall: 0.27900 F1: 0.27810 (that's pretty ok)
- ‘exercised_stock_options’, ‘total_stock_value’, ‘salary’, ‘bonus’: Precision: 0.33352 Recall: 0.30550 F1: 0.31889
- ‘exercise_stock_options’, ‘total_stock_value’, ‘salary’, ‘bonus’, ‘shared_receipt_with_poi’: Precision: 0.32256 Recall: 0.30950 F1: 0.31590 (so ”shared receipt” actually makes things worse...)

- (All seven most promising features that I chose earlier by hand) 'shared_receipt_with_poi", 'from_this_person_to_poi', 'from_poi_to_this_person', 'long_term_incentive', "salary", 'exercised_stock_options', "bonus": Precision: 0.30445 Recall: 0.26350 F1: 0.28250

So the best four features seem to be ['exercised_stock_options', 'total_stock_value', 'salary', 'bonus'].

5 Creating Composite Features

I tried to combine ['shared_receipt_with_poi', 'from_poi_to_this_person'] because even though these features seem very correlated, there is a region in the plot with no POIs at all. So, I thought that if we could radially isolate POIs from non-POIs - that could be a decision boundary for a decision tree. So I tried to radially combine these two features ($x^2 + y^2$) - but this feature actually performs poorly compared to "shared receipt with poi". The SelectKBest score for this feature is 1.7587. I use the best decision tree to test this feature, and get:

Accuracy: 0.81333 Precision: 0.25000 Recall: 0.20000 F1: 0.22222 F2: 0.20833

Total predictions: 75 True positives: 2 False positives: 6 False negatives: 8 True negatives: 59

Without this new feature and 7 best features from the previous section, the simple decision tree performs better, with a precision of 0.25 and recall of 0.3 with a 5 fold stratified shuffle split cross validator.

From the plot (compare_three_features(["poi","bonus", "total_payments", 'shared_receipt_with_poi'], True, "radial")), it looks like if I **scale these features and simply combine them radially**, then a simple decision tree would be enough to separate out POIs from non-POIs.

new_dict=combine_features(["bonus", "total_payments", 'shared_receipt_with_poi'], "best_3", True, False, "radial")

And running a simple decision tree with the two new features:

Accuracy: 0.81273 Precision: 0.27763 Recall: 0.25250 F1: 0.26447 F2: 0.25715

Total predictions: 15000 True positives: 505 False positives: 1314 False negatives: 1495 True negatives: 11686

6 Validation

We want our algorithm to be able to classify people as POIs and non-POIs for cases in which we do not know the outcome yet. To test the strength of predictions made by our algorithm, we need to test it on a blind sample. For instance, we let the code learn by studying features of cases for which we know the final classification, but we should set aside part of this "known" sample to check how well the code learned.

One way to do it is to randomly split some percentage of the known sample, use the larger chunk for training and the smaller chunk for testing. For cases where there is a nearly even number of data points for each label, this is a reasonable choice. But if there is a disproportionately higher fraction of data points in one class, it is better to stratify the training and testing sets so that both sets have the same percentage of data points from each class.

Another way that maximizes the use of the data is cross-validation. For instance, a Kfold cross validation will split the known data in k-bins. Then, it would train and test the data k times - each time using (1/k)-th fraction (1 bin) for testing and the rest (k-1 bins) for training. The final evaluation metric (accuracy/precision/recall) uses an average of the k runs to evaluate the algorithm. In this way, cross-validation allows us to use the whole dataset for training and then for testing. This is useful in the Enron case because there are so few POIs, it would be good to be able to minimize train-test data loss by using a cross-validation method. I used precision and recall for validation rather than accuracy values because the disproportionate distribution between the two classes means even if there are a lot of false negatives in the test set, the accuracy would be high. Precision = Number of true positives / (true positives + false positives) and Recall = Number of true positives / (true positives + false negatives).

For this project, I have explored Kfold, ShuffleSplit, and chose stratified shuffle split because of the ratio of POIs to non-POIs in this problem is so low, it is important for the training and testing sets to have an equal sample of both classes. One way to do this is to use train test split with stratification - which is what I do before running SelectKBest. But for actual testing of algorithms, I used the cross validation code that was provided in tester.py because Stratified Shuffle Split was set up perfectly.

However, while tuning parameters using GridSearchCV, I modified the test_classifier function from tester.py and wrote grid_search_test_classifier(), to see the best fit parameters for each fold (the parameters for gridsearchcv changed each time the data was trained). I took the median values for these parameters, and tweaked a few parameters around the median values to arrive at the best fit parameters. **The complete GridSearchCV process can be run by uncommenting automated_fitting() in line 464 - but all the training and testing takes a while.**

7 Picking an Algorithm and Tuning Parameters

I tried Decision Tree, Adaboost, Support Vector Machine, K Nearest Neighbors, RandomForest, Naive-Bayes, Multi-Layer Perceptron and Extra Trees. Some of these algorithms, such as SVM, K Nearest Neighbors did not do well when I tried to tune parameters and select features by hand.

For each algorithm, we have to choose parameter values carefully. The problem with choosing the wrong parameters is that if we choose the wrong optimizer or kernel, or use the wrong value of C/alpha/gamma, the code can either take a long time, or prioritize the wrong term in the algorithm. For instance, in decision tree, if the min sample split is too small, then the code will overfit, and we will get terrible results with the test sample even if it does well with the training sample. If we choose a "linear" kernel for a SVC where the label depends on sum of square of the parameters, then we will get very poor results from SVM, but possibly a very good approximation with 'rbf'. So, we may have the correct algorithm but choosing the wrong parameters will give us vastly different results.

For decision tree, I tuned (by hand) max_depth (number of branches), min_sample_split, and finally used gridsearchcv to choose a min_sample_split and whether or not to presort the labels before fitting. For MLPclassifier, I tuned the solver and alpha values. What a solver exactly is is still a blackbox to me - but I understand what optimization is. Solver for MLP seems analogous to entropy for decision tree - I decided on the sgd solver in the hand tuning phase. For Adaboost I only tweaked the number of weak estimators. I also changed random state (which is just the seed for random number generator) - I don't think it is particularly important. For SVC, I found that changing C beyond 1 takes a really long time, and gamma did not have a big impact. I used both linear and rbf kernels but didn't get a good result with either. There are no parameters for naive bayes.

An interesting effect of parameter tuning with my best algorithm: I tried to use the parameter class weight as suggested by a reviewer when running random forest, and assigned a weights of 0.7 to poi and 0.3 to non-poi. This resulted in: Accuracy: 0.83562 Precision: 0.45566 Recall: 0.35200 F1: 0.39718 F2: 0.36878

Then I changed weights to poi: 0.85 an non-poi: 0.15 and got: Accuracy: 0.82385 Precision: 0.41971 Recall: 0.37900 F1: 0.39832 F2: 0.38650

As expected, precision gets worse but recall gets better as we increase the weight of poi class. But changing the relative weights any more (that is, prioritizing poi any more over non-poi class) starts to make the results worse (I think prioritizing poi too much turns it into a high bias code - as explained in the lectures).

Besides parameter tuning, another important factor is whether or not to scale features. Decision trees, adaboost, random forest/extra trees are not sensitive to features scaling, but neural networks, knearestneighbors and SVM are. I used minmaxscaler for the latter three, but in the end, I decided against using these algorithms, so I do not use scaling in the final algorithm. I also allow the option of scaling parameters when trying to combine features using the function combine_features(). While combining features that are orders of magnitudes apart, the bigger feature will dominate, so scaling is important.

MLP, K nearest neighbors were taking too long, so in the end, I selected Decision Tree, Adaboost and Randomforest as the best algorithms to try and fine tuned them with the help of cross-validation algorithm (1000 fold stratified shuffle split) and GridSearchCV.

I tuned the parameters by using a pipeline and gridsearchCV with PCA and each algorithm. Using 1000 fold stratified shuffle split from the tester functions, I noted which parameter values occur most often, tested just the median values, and then slight variations, and arrived at just one combination of parameters for each algorithm. The best results I get are as follows:

	Decision Tree	Adaboost	Random Forest
Precision	0.45235	0.32889	0.43180
Recall	0.38450	0.18500	0.37200
F1	0.41568	0.23680	0.39968

So both decision tree and random forest perform approximately at the same level with the four best features [‘exercised_stock_options’, ‘total_stock_value’, ‘salary’, ‘bonus’].

Using the two features I constructed, along with all the other features, I get:

	Decision Tree	Adaboost	Random Forest
Precision	0.278	0.264	0.25309
Recall	0.253	0.133	0.22550
F1	0.264	0.177	0.23850

So these features do not pass the test.

8 Conclusion

Looking at the results from the previous section, I decided that **DecisionTree with these four features: (‘exercised_stock_options’, ‘total_stock_value’, ‘salary’, ‘bonus’) should be the final algorithm**. Please refer to previous section for a comparison of results from DecisionTree, AdaBoostand RandomForest.

9 References

I used sklearn documentation to learn to use the pipeline and cross validation, as well as to use the new algorithms (RandomForest/Extra Trees) that are not covered in the course. I used stackoverflow to solve python issues.

10 Appendix: List of Functions in Code

Most of the parameters for the functions are self explanatory, and I provided some docstrings and examples to help out.

- find_by_name(): has one required parameter - the name of a person in the dataset as it appears in the dictionary key.
- find_an_interesting_person(): requires a feature name (fieldname) and a limit. Prints out the names of all the people who lies above that limit for that feature. For instance to find everyone with a salary above 1000000, use: find_an_interesting_person(“salary”, 1000000)
- find_outliers(): take a feature name as the only parameter. This function is described in § 2.
- compare_three_features(): Takes in a list of features to make a 3D interactive plot. Scales features (optional), takes in upper axis limits for each feature (in respective order). This plot can either only show a training set or all the available features. ts_size allows us to vary test fraction size.
- combine_features (): This is an attempt to make a composite feature by either combining them radially = $x^2 + y^2$ or linearly = x + y Any number of things can be combined this way, and a new dictionary is returned with the dataset added. The first parameter is a list of features to be combined, the second parameter is the name of the new composite feature. If scaled is True, each field is scaled before combining. ignore_nan set to True means even if one field has NaN value, the others are combined, assuming the NaN is 0. Not ignoring nan (ignore_nan = False) means we just set the new feature to 0 if any of the component featutes is 0.
- find_best_features(): Runs selectkbest on a dataset (the main dataset by default, but also can take in a modified dataset with a new feature), and list of features to rank.

- simple_decision_tree(): Tests a new feature using a simple decision tree. This has very limited use as I made it specifically for the composite features that I created - takes in a dataset and a list of features to use in the decision tree.
- grid_search_test_classifier(): This is just a copy of classifier test code - with a small addition - it let's us see what are the best parameters if we are using it with gridsearchCV. Also, I use only 10 folds.
- automated_fitting(): Using pipelines of a selection of algorithms+PCA, GridSearchCV allows us to see what the different parameters turn out to be for each fold of cross validation (using stratified shuffle split). Then in the next function (best_automated_classifier()), I use a median of these parameters, and tweak a little by hand to arrive at the best algorithm.
- best_automated_classifier(): Does not require any parameter, but can accept a distionary with modified/new features and list of feature names to use. There are three pipelines set up, random forest, adaboost and decision tree. The default algorithm (first argument for this function) is random forest. The default dataset is the initial dataset and the default list of features is the seven best features I determined by SelectKBest + testing by hand.